Identifying Failure-Inducing Combinations Using Tuple Relationship

Xintao Niu[#1], Changhai Nie[#2], Yu Lei[†3], Alvin TS Chan[*4]

[#] *State Key Laboratory for Novel Software Technology, Nanjing University, China, 210093*

[1]niuxintao@gmail.com; [2]changhainie@nju.edu.cn

[†] *Department of Computer Science and Engineering, the University of Texas at Arlington, USA*

[3] ylei@cse.uta.edu

[*] *Department of computing, Hong Kong Polytechnic University*

[4] cstschan@comp.polyu.edu.hk

***Abstract*—Combinatorial testing (CT) aims at detecting interaction failures between parameters in a system. Identifying the failure-inducing combinations of a failing test configuration can help developers find the cause of this failure. However, most studies in CT focus on detecting the failures rather than identifying failure-inducing combinations. In this paper, we propose the notion of a tuple relationship tree (TRT) to describe the relationships among all the candidate parameter interactions. TRT reduces additional test configurations that need to be generated in the fault localization process, and it also provides a clear view of all possible candidate interactions. As a result, our approach will not omit any possible interaction that could be the cause of a failure. In particular, we can identify multiple failure-inducing combinations that overlap with each other. Moreover, we extend our approach to handle the case where additional failure-inducing combinations may be introduced by newly generated test configurations.**

***Index Terms*—Combinatorial testing, Fault localization, Tuple relationship tree, overlapping combinations, New import combinations.**

I. INTRODUCTION

The behavior of a software system may be influenced by many factors, such as input parameters and configuration options. Some interactions of these factors may cause failures that may be difficult to track. One way to ensure software correctness is to perform exhaustive testing of all combinations of these factors. This is, however, not feasible when the number of factors is large. Combinatorial testing (CT) is an efficient testing method to deal with this problem [1]. It has been shown to be effective at detecting failures caused by interactions of parameters. However, most work on CT has primarily focused on detecting failures. Little work has been reported that provides support for identifying failure-inducing combinations, i.e., combinations that cause the failures that are detected by CT [2].

Generally, when a test case with a configuration failed during the system testing, there are $2^n - 1$ (n is the number of parameters of the configuration) possible combinations that may cause this failure. Identifying which specific combinations that constitutes to failure-inducing combinations is an important problem in that these combinations can help to reduce the scope of the code that needs to be inspected to locate the actual fault.

Recently, several methods have been proposed that attempt to identify failure-inducing combinations. However, there is much room left to be improved in handling the following two major aspects: 1. Identifying multiple combinations which overlap with each other. 2. Identifying additional failure-inducing combinations that are introduced by newly generated test configurations.

To identify the failure-inducing combinations in a failing test configuration, we proposed an approach based on the notion of a tuple relationship tree (TRT). In our approach, we first generate a tree structure to capture the relationship between different combinations in a failing test configuration. Then we select an unknown combination that has yet to be determined whether it is a failure-inducing combination. After a combination has been selected, we analyze it to confirm if it is a failure-inducing combination. This is followed by progressive checks on other related combinations. This process continues until no other unknown combination could be found.

As we know, the TRT provide a clear view of all possible interactions in a failing test configuration. As a result, our approach will not omit any possible combination that may induce the cause of a failure. In particular, we can identify multiple failure-inducing combinations that overlap with each other. Our approach is shown to be efficient, which requires only a few number of extra test configurations to identify failure-inducing combinations. The efficiency of our approach lies in the combination selecting strategy, i.e., the order of selecting the unknown combination from the remaining unknown combinations. The main idea behind our selecting strategy is the following: 1. Greedy step: we select a sequence of closely related unknown combinations from the remaining unknown combinations and sort them according to their relationships with each other. Generally, there is more than one such sequence. As such, we will select the one with the maximum number of combinations for that we can determine the combinations as much as possible in the iteration. 2. Search step: for the sequence we select in the first step, we use binary search to find the combination in the sequence as the one to be analyzed for this turn. Since the combinations in the sequence are closely related and sorted according to their relationships

with each other, we can largely reduce the number of extra test configurations needed to identify the failure-inducing combinations through the use of binary search technique. Furthermore, we proposed an augmented version of our approach which reinforces the process of determining a combination to see if it is a failure-inducing combination. Such an improvement provides a preliminary solution to handle the case where additional failure-inducing combinations may be introduced by newly generated test configurations.

Contribution of this paper: 1) We propose a new approach to identifying the failure-inducing combinations in a failing test configuration; 2) Our approach can identify multiple, overlapping combinations; 3) The augmented version of our approach can handle the case where additional failure-inducing combinations may be introduced by newly generated test configurations.

The rest of this paper is organized as follows: Section 2 introduces some preliminary definitions and propositions. Section 3 describes our model for identifying failure-inducing combinations. Section 4 presents our experimental results. Section 5 summarizes related work. Section 6 provides concluding remarks and discusses the future work.

II. PRELIMINARY

Assume that the SUT (software under test) has *n* parameters, and each parameter c_i has a_i discrete values from the finite set V_i, i.e., $a_i = |V_i|(i = 1,2,..n)$. Some of the definitions and propositions below are originally defined in [3].

Definition 1 A *test configuration* is an array of *n* values, one for each parameter of the SUT, which is denoted as (v_1, v_2… v_n), where $v_1 \in V_1$, $v_2 \in V_2$… $v_n \in V_n$.

For example, consider a web application which may be influenced by various aspects including operating system, network bandwidth, browser and codec. We list the possible values of each parameter in Table I. Then (Mac OS, 50M, Chrome, AC-3 ACM) is a test configuration for the application.

TABLE I. PARAMETER VALUES OF THE WEB APPLICATION

OS	Bandwidth	Browser	Codec
Windows	50M	Chrome	AC3Filter
Mac OS	200M	FireFox	MPEG Layer-3
Ubuntu	1G	Opera	AC-3 ACM

Definition 2 A *tuple* is also an array of some values, one for a parameter of the SUT. We denote a tuple as [-,…, v_{n_1} ,-,…v_{n_k} ,…] which means the tuple consists of *k* values that comes from the parameter c_{n_1} ,c_{n_2} ,…c_{n_k} of the SUT, and the '-' means the excluded corresponding parameter value. We called the number of values in a tuple the *size* of the tuple. We also call a tuple of size *k* the *k-size* tuple. The tuple is also known as the combination mentioned before.

For example, [Mac OS, 50 M, -, -] is a 2-size tuple for the web application.

Definition 3 A tuple is called a *faulty tuple*, when every possible test configuration containing this tuple results in a failure. And a tuple is called a *healthy tuple* when we find at least one passed test configuration that contains this tuple.

For example, if [Mac OS , 50 M , -, -] is a faulty tuple, then all the test configurations containing it, such as (Mac OS , 50 M , Chrome, AC-3 ACM), (Mac OS , 50 M , Firefox, AC3Filter), will fail. Conversely, if there is at least one of these test configurations that passes, [Mac OS, 50 M, -, -] is labeled as a healthy tuple.

Definition 4 For two tuples A and B, if every parameter value in A are also in B, and the size of A is less than B, then we called A the *child* of B, and B the *parent* of A.

Furthermore, if the size of A is exactly one less than B, then the relationship between A and B are *direct*.

For example [Mac OS, 50 M, Chrome, -] is the parent of [-, 50M, -, -], and [Mac OS, 50 M, Chrome, -] is the direct parent of [Mac OS, 50 M, -, -].

Definition 5 If a tuple is a *faulty tuple* and all its child tuples are *healthy tuple*s, we then call the tuple *a minimal faulty tuple*.

Identifying *minimal faulty tuples* can facilitate debugging effort, as it can reduce the scope of the code that needs to be inspected.

A. Propositions

We list four propositions. Due to space limitation, we will skip the proofs of these propositions as they are pretty straightforward.

Proposition 1 All the tuples in a passed test configuration are healthy tuples.

Proposition 2 If t_a is the parent of t_b , t_b is the parent of t_c , then t_a is the parent of t_c.

Proposition 3 All the parent tuples of a faulty tuple are faulty tuples.

Proposition 4 All the child tuples of a healthy tuple are healthy tuples.

Note: these definitions and propositions give us an ideal framework to identify the minimal faulty tuple, which assume that the SUT is deterministic software and a test configuration contains a faulty tuple must result in a failure during testing.

III. FAILURE-INDUCING COMBINATIONS IDENTIFYING MODEL

We will introduce our failure-inducing identifying model with an example SUT. The SUT consists of 4 parameters, each having 3 values. We generated a 2-way covering array to test this SUT. The 2-way covering array and the test results are listed in Table II.

TABLE II. EXECUTED TEST CONFIGURATIONS

No.	Test configuration				Result
1	1	1	1	1	Pass
2	1	2	2	2	Pass
3	1	3	3	3	Pass
4	2	1	2	3	Pass
5	2	2	3	1	Fail
6	2	3	1	2	Pass
7	3	1	3	2	Pass
8	3	2	1	3	Pass

No.	Test configuration				Result
9	3	3	2	1	Pass

A. Constructing TRT

First, we construct a TRT for a failing test configuration. A TRT is a tree in which each node represents a tuple in the failing test configuration and each edge represents a direct parent and child relation from one node to another node.

For instance, for the failing configuration (2, 2, 3, 1) in Table II, the TRT is shown in figure 1.

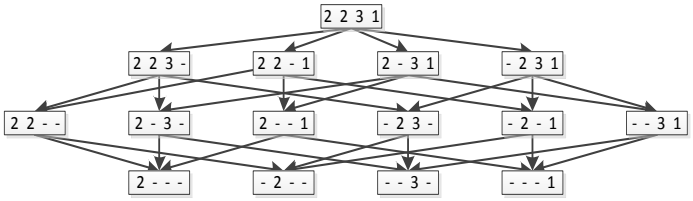

Fig. 1. TRT for (2,2,3,1)

B. Initial known tuples

In fact, some tuples in a TRT can be easily determined to be faulty tuple or healthy tuple from the results of the executed test configurations. Thus we do not need to generate extra test configurations to analyze them. We can determine the type of a tuple using static review as follows:

First, the root tuple in a TRT must be a faulty tuple. This is because all the possible test configurations contain the root tuple is original configuration, which failed during testing. So it is a faulty tuple by definition. We mark the root tuple as a faulty tuple. Second, tuples that appear in one or more passed test configurations are healthy tuples by definition. We mark such tuples as healthy tuples.

After the above steps, the TRT evolves as shown in Figure 2, where dark nodes represent faulty tuples, grey nodes represent healthy tuples, and white nodes represent unknown tuples.

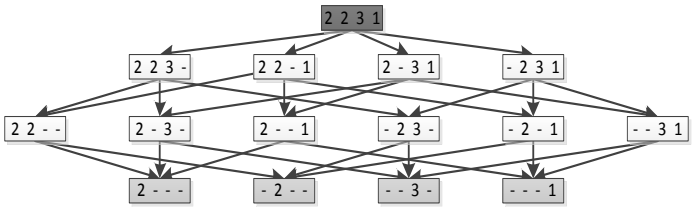

Fig. 2. TRT after initial process

C. Identifying process

Having derived the TRT, there still remain many unknown tuples. We need to determine which class they belong to respectively, i.e., (faulty tuple or healthy tuple) to identify the minimal faulty tuples. We will make the following assumption to facilitate the process of determining unknown tuples in the TRT:

Assumption: The generated extra test configuration will not introduce new faulty tuples. (This assumption may not always be true. This will be discussed further in the next section.)

Based on this assumption, we will get the following lemma.

Lemma 1: For a tuple, we generate an extra test configuration that contains this tuple. If the extra test configuration passes, then this tuple is a healthy tuple. If the extra test configuration fails, then the tuple is a faulty tuple.

Proof. According to Definition 2, it is obvious that this tuple is a healthy tuple when the extra test configuration passes.

When the extra configuration fails, this is a faulty tuple (or there exists no faulty tuple and this test configuration would not fail because the assumption says that this extra configuration will not introduce new faulty tuples). ▮

With this lemma, the process of determining all the unknown tuples is also very straightforward that we will just take the followed example to illustrate it. Consider the TRT in Fig 2. the remaining unknown tuples are: [2,2,3,-],[2,2,-,1],[2,-,3,1],[-,2,3,1],[2,2,-,-],[2,-,3,-],[2,-,-,1], [-,2,3,-],[-,2,-,1],[-,-,3,1]. We first select the tuple [2,2,3,-] to be analyzed (the order in which tuples are selected from the remaining unknown tuples will be discussed later). We generate an extra test configuration (2,2,3,2) for the tuple [2,2,3,-]. Let's assume that this test fails during testing, which we subsequently label this tuple as a faulty tuple. We also label all its parent tuples to be faulty tuples. However, it has only one parent tuple in the TRT, which is [2,2,3,1] and has been labeled to be a faulty tuple in the initial step. So we skip this step. Next we select the tuple [2,2,-,-], and find this tuple is a healthy tuple after we generate a test configuration (2,2,1,2) which is assumed to pass during testing. We also label all its child tuples to be healthy tuples. This process continues until there are no unknown tuples.

TABLE III. IDENTIFYING EXAMPLE

Iteration	Selected tuple	Extra Test	Result	Additional tuples
1	[2,2,3,-]	(2,2,3,2)	faulty	-
2	[2,2,-,-]	(2,2,1,2)	healthy	-
3	[2,2,-,1]	(2,2,1,1)	healthy	[2,-,-,1], [-,2,-,1]
4	[2,-,3,1]	(2,1,3,1)	faulty	-
5	[2,-,3,-]	(2,1,3,2)	faulty	-
6	[-,2,3,1]	(1,2,3,1)	faulty	-
7	[-,2,3,-]	(1,2,3,2)	healthy	-
8	[-,-,3,1]	(1,1,3,1)	healthy	-

The details of our selection process are listed in table III. Column "Selected tuple" shows the tuple selected for this iteration, column "Extra Test" shows the extra test configuration that contains the selected tuple. Column "Result" shows our analysis result, i.e., whether the selected tuple is a healthy or faulty tuple. The last column "Additional tuples" shows the additional parent or child tuples of this selected tuple which can be determined according to Propositions 3 and 4.

Fig. 3. TRT after identifying process

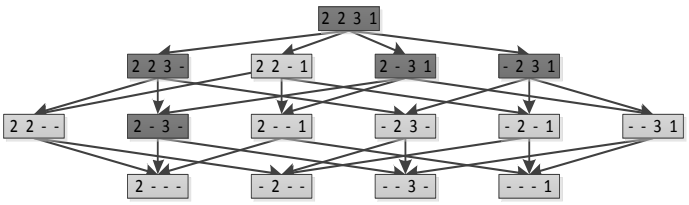

Figure 3 shows the updated TRT. As all the unknown tuples in the TRT are determined, we can easily find the minimal faulty tuples according to Definition 4, i.e. [2,-,3,-] and [-,2,3,1].

D. Tuple Selection strategy

It is important to note that the selection of an unknown tuple from the remaining unknown tuples has a significant impact on the efficiency of our approach. This is particularly so when there are still many unknown tuples to be analyzed in the TRT. Our selection strategy aims at minimizing the number of extra test configurations. Before we describe this strategy, we present the following definitions:

Definition 6 A *path* is a sequence of tuples in a TRT in which every tuple is the direct parent of the tuple that follows.

For example, [2,2,3,1]→[2,2,3,-]→[2,-,3,-]→[2,-,-,-] is a path.

Moreover, a path is said to be an *unknown path* if every tuple in this path is an unknown tuple. A path is said to be the *longest unknown path* when this path is an unknown path that has the maximum number of tuples in the TRT.

The selection strategy is described in Algorithm 1. In this algorithm, variable *lastChosen* represents the last chosen tuple, while the *currentPath* stores a path of the TRT. The variables *head*, *middle*, and *tail* are the indexes of the tuples in *currentPath*. All these variables are member variables owned by the TRT.

Our selecting strategy has two main steps: Greedy step and binary search step. For the first step, we will choose a longest unknown path from the TRT. The idea behind this step is to enable us to determine as many unknown tuples as possible. There are two conditions that we need to apply in order to determine the longest unknown path, i.e., reset the *currentPath* (line 9): 1) The last chosen tuple is null, which means our select function is invoked for the first time; 2) The index head is greater than tail, which means that all the tuples in the current path have been determined. After getting a new longest unknown path (line 9), our select function let *head* point to the first tuple of path and *tail* point to the last tuple of path (line 10 ~11). Furthermore, we let the *middle* index point to the first tuple of path (line 10). The reason why we first let the middle point to the first tuple of the path is that we can determine as soon as possible if the path has faulty tuples. For example, consider a path [1,1,1,1,-]→[1,1,1,-,-]→[-,1,1,-,-]→[-,1,-,-,-]. We first let *middle* point to [1,1,1,1,-]. If it is a healthy tuple, then we immediately determine that there are no faulty tuples in this path.

For the second main step, we choose binary search to get a tuple to be tested from the *currentPath*. As we know, the tuples in the *currentPath* is sorted according to their relationships, i.e., every tuple is the direct parent of the tuple that follows. Such that if we have determined one tuple in the path, the tuples before this selected tuple or after this selected tuple will also be determined. So we use binary search to choose this tuple so that we can minimize the number of extra test configurations (O ($\log_2$ n)) that is used to determine all the tuples in the *currentPath*. The details are as follow: we first determine if the last chosen tuple is null (line 2). If not, we change the index of *tail* or *head* to ensure the tuples between *tail* and *head* are all unknown tuples. When the last tuple is a healthy tuple, all the subsequent tuples are healthy tuples. Thus the process will let *tail* point to the tuple before *middle* (line 3 ~ 4). If the last tuple is a faulty tuple, then the tuples before it are faulty tuples. So the process will let the *head* point to the tuple after the *middle* (line 5 ~ 6). Based on the binary search method, set the *middle* index to point to the middle tuple that is between the *tail* and *head* (line 7). At the end, we return the tuple which *middle* points to in the *currentPath* and set it to be the last chosen tuple for the next iteration (lines 12~13).

Algorithm 1 select strategy

```
1:          define select():
2:              if lastChosen != null:
3:                  if lastChosen.isHealthy:
4:                      tail = middle – 1
5:                  elseif lastChosen.isFaulty:
6:                      head = middle+1
7:                  middle = ( head + tail )/2
8:              if lastChosen == null || tail < head:
9:                  currentPath = get_longest_path()
10:                 head = middle = 0
11:                 tail = currentPath.size – 1
12:             lastChosen = currentPath[middle]
13:             return lastChosen
```

The following example shows how our strategy works:

Assume that we have a failing configuration (1, 1, 1, 1, 1, 1, 1, 1). Also assume that we do not have any information about other test configurations. We construct the TRT for this configuration, and initialize the root tuple to be a faulty tuple.

Next, our selection process starts. We get a longest unknown path first:

[1 , 1 , 1 , 1 , 1 , 1 , 1 , -]→[1 , 1 , 1 , 1 , 1 , 1 , - , -]→[1 , 1 , 1 , 1 , 1 , - , - , -]→[1 , 1 , 1 , 1 , - , - , - , -]→[1 , 1 , 1 , - , - , - , - , -]→[1 , 1 , - , - , - , - , - , -]→[1 , - , - , - , - , - , - , -].

And then we choose the first tuple: [1 , 1 , 1 , 1 , 1 , 1 , 1 , -] to be analyzed. We generated an extra test configuration (1 , 1 , 1 , 1 , 1 , 1 , 1 , 0), which is assumed to fail. So we label this tuple to be a faulty tuple. We also label all its parent tuples to be faulty tuples. For this particular tuple, the only parent tuple is the root tuple [1 , 1 , 1 , 1 , 1 , 1 , 1 , 1], which is already labeled to be a faulty tuple.

As the last chosen tuple is a faulty tuple, then we select the tuple [1 , 1 , 1 , 1 , - , - , - , -], We generate an extra configuration (1 , 1 , 1 , 1 , 0 , 0 , 0 , 0), which is assumed to pass. We label this tuple and all its child tuples to be healthy tuples, which include the following tuples: [1 , 1 , 1 , - , - , - , - , -] , [1 , 1 , - , - , - , - , - , -] ,[1 , - , - , - , - , - , - , -] in the path.

The next tuple chosen is [1 , 1 , 1 , 1 , 1 , 1 , - , -]. As the last chosen tuple is a healthy tuple. It is also determined to be a healthy tuple. We label this tuple and all its child tuples to be healthy tuples (including the last unknown tuple in this path: [1 , 1 , 1 , 1 , 1 , - , - , -]). As all the tuples in the path are determined, we get a new longest unknown path. If there are more than one longest unknown path, we select one of them arbitrarily. In this example, we get the following longest unknown path:

[1 , 1 , 1 , 1 , 1 , - , 1 , 1]→[1 , 1 , 1 , 1 , 1 , - , 1 , -]→[1 , 1 , 1 , 1 , - , - , 1 , -]→[1 , 1 , 1 , - , - , - , 1 , -]→[1 , 1 , - , - , - , - , 1 , -]→[1 , - , - , - , - , - , 1 , -]→[- , - , - , - , - , - , 1 , -].

We repeat the above process until all the tuples in the TRT are all determined. Our complete process for this example is listed in table IV. At the end of this process, we find the minimal faulty tuple is the tuple [1 , - , - , - , - , - , 1 , -].

TABLE IV. SELECTING EXAMPLE

Path	[1 , 1 , 1 , 1 , 1 , 1 , 1 , -]→[1 , 1 , 1 , 1 , 1 , 1 , - , -]→[1 , 1 , 1 , 1 , 1 , - , - , -]→[1 , 1 , 1 , 1 , - , - , - , -]→[1 , 1 , 1 , - , - , - , - , -]→[1 , 1 , - , - , - , - , - , -]→[1 , - , - , - , - , - , - , -].	
No.	**Selecting tuple**	**Result**
1	[1 , 1 , 1 , 1 , 1 , 1 , 1 , -]	Faulty
2	[1 , 1 , 1 , 1 , - , - , - , -]	Healthy
3	[1 , 1 , 1 , 1 , 1 , 1 , - , -]	Healthy
Path	[1 , 1 , 1 , 1 , 1 , 1 , - , 1 , 1]→[1 , 1 , 1 , 1 , 1 , 1 , - , 1 , -]→[1 , 1 , 1 , 1 , - , - , 1 , -]→[1 , 1 , 1 , - , - , - , 1 , -]→[1 , 1 , - , - , - , - , 1 , -]→[1 , - , - , - , - , - , 1 , -]→[- , - , - , - , - , - , 1 , -].	
No.	**Selecting tuple**	**Result**
4	[1 , 1 , 1 , 1 , 1 , - , 1 , 1]	Faulty
5	[1 , 1 , 1 , - , - , - , 1 , -]	Faulty
6	[1 , - , - , - , - , - , 1 , -]	Faulty
7	[- , - , - , - , - , - , 1 , -]	Healthy
Path	[1 , 1 , 1 , 1 , 1 , 1 , - , 1]→[1 , 1 , 1 , 1 , 1 , 1 , - , - , 1]→[1 , 1 , 1 , 1 , - , - , - , 1]→[1 , 1 , 1 , - , - , - , - , 1] →[1 , 1 , - , - , - , - , - , 1]→[1 , - , - , - , - , - , - , 1]→[- , - , - , - , - , - , - , 1]	
No.	**Selecting tuple**	**Result**
8	[1 , 1 , 1 , 1 , 1 , 1 , - , 1]	Healthy
Path	[- , 1 , 1 , 1 , 1 , 1 , 1 , 1]→[- , 1 , 1 , 1 , 1 , 1 , 1 , -]→[- , 1 , 1 , 1 , 1 , - , - , 1]→[- , 1 , 1 , - , - , - , 1 , -]→[- , 1 , - , - , - , - , 1 , -]→[- , 1 , 1 , - , - , - , - , -]	
No.	**Selecting tuple**	**Result**
9	[- , 1 , 1 , 1 , 1 , 1 , 1 , 1]	Healthy

E. Removal of the assumption

By far our identifying process is based on the fundamental assumption that the generated extra test configuration will not introduce new faulty tuples. Since this assumption is not always true, we present a preliminary solution to eradicate such an assumption in this section. For ease of reference, we refer to the assumption as "TRT assumption".

As we know, in our previously approach, we just use one extra test configuration to test the selected tuple. We can see that if the extra test configuration passes, this tuple is surely a healthy tuple, but if it fails, this tuple is not deterministic a faulty tuple. This is because based on definition only when we make sure every possible test configurations which contain the selected tuple will fail can we determine this tuple is a faulty tuple. However, having an exhaustive testing of all these test configurations is too costly and not practical. So we should make a tradeoff by restricting the number of extra test configurations for determining a faulty tuple. It is obviously that the more the number of test configurations for determining a faulty tuple, the higher probability that a tuple may be faulty, which lead to higher cost in identifying failure-inducing tuples. In the extreme case, a tuple is deterministic to be a faulty tuple when this value is equal to the number of the possible test configurations contain this tuple. Conversely, the less the number of test configurations for determining a faulty tuple, the less probability that this tuple is a faulty tuple, which leads to less cost needed to identify failure-inducing tuples. Notably, the extreme case of this situation is that we only take one extra test configuration to determine a faulty tuple, which turns into our previously algorithm. Due to space limitation, we will not discuss how to determine the best number of test configurations for a SUT to determine a faulty tuple in this paper.

Our augmented approach without the TRT assumption is shown in Algorithm 2. Similar to the algorithm with the TRT assumption in place, this approach also needs to determine whether there exists any tuple which is still unknown in the TRT (line 2), select a tuple from the unknown tuples (line 3), generate and execute an extra test configuration that contains this tuple (line 5 ~6). The difference is that this approach uses two new variables: *num_needed* and *tempSuite*. The former indicates the number of extra test configurations needed to determine a faulty tuple and the latter records the failing extra test configurations generated for the current tuple that need to be analyzed.

If the current extra test configuration fails (line 7), we add this test configuration in the *tempSuite* (line 8). Then we will evaluate whether the number of test configurations in *tempSuite* is less than *num_needed*. If so, generate a new test configuration and execute it for the next iteration (line 10~11). Otherwise, we break the loop (line 13). If the number of test configurations in the *tempSuite* is less than *num_needed* , which means that this tuple is a healthy tuple, we will set this tuple and all its child tuples to be healthy tuples (line 15 ~17). Furthermore, although the tuple may be considered as a healthy tuple, some extra test configurations recorded in the *tempSuite* may fail during testing. As a result, it indicate that these test configurations introduced new faulty tuples (otherwise, there will exist no faulty tuple, and these test configurations would pass). We recursively use this approach to analyze the failing test configurations in the *tempSuite* to find these introduced failure-inducing tuples (line 18~19). If the number of test configurations in *tempSuite* is greater than *num_needed*, which means that this tuple is a faulty tuple, we will set this tuple and all its parent tuples to be faulty tuples (line 21 ~23).

We present a simple example to show how our identification approach without TRT assumption works:

For a failing test configuration (1,1,1), let *num_needed* = 3. We first consider the tuple [1,1,-]. We generate a test configuration (1,1,2), which is assumed to fail. We continue to generate another test configuration (1,1,3), which is also assumed to fail. As the number of test configurations in *tempSuite* is still less than *num_needed*, so we continue to generate the next test configuration (1,1,4), and assume that it fails again. Now the number of test configurations in *tempSuite* is 3, which is equal to *num_needed*. This means that this tuple is a faulty tuple. We repeat our process until there are no unknown tuples in the TRT.

Algorithm 2 Identifying process without TRT assumption

```
define identify_NA(TRT):
    while TRT.unknown_tuples != ∅:
        tuple = TRT.select(unknown_tuples)
        tempSuite = []
        test_config = gen_extra(tuple)
        result = exec(test_config)
        while result == fail :
            tempSuite.append(test_config)
            if numberOf(tempSuite) < num_needed :
                test_config = gen_extra(tuple)
                result = exec(test_config)
            else :
                break
        if numberOf(tempSuite) < num_needed :
            TRT.set(tuple,healthy)
            for each_child of tuple:
                TRT.set(each_child, healthy)
            for each_test_config of tempSuite:
                identify_NA(TRT(each_test_config))
        else :
            TRT.set(tuple,faulty)
            for each_father of tuple:
                TRT.set(each_father, faulty)
```

It is important to note that when the tuple is evaluated to be a healthy tuple, but some extra test configurations containing this tuple fail during testing, we should recursively identify these extra test configurations to find new failure-inducing tuples. For example, when we analyze the tuple [1,-,1],we generate two configurations (1,2,1), (1,3,1). Assume that the former fails and the later passes. As (1,3,1) passes, [1,-,1] is a healthy tuple, and then we should recursively analyze configuration (1,2,1) to find new failure-inducing tuples.

Our complete identifying process is listed in table V. The minimal faulty tuples are [1,1,-],[-2,1],[2,1,-] and [3,1,-].

TABLE V. WITHOUT TRT ASSUMPTION EXAMPLE

test configuration (1,1,1)				
No.	**Tuple Select**	**Generate Configurations**		
1	[1,1,-]	(1,1,2) fail	(1,1,3) fail	(1,1,4) fail
2	[1,-,-]	(1,2,2) pass	-	-
3	[1,-,1]	(1,2,1) fail	(1,3,1) pass	-
4	[-,1,1]	(2,1,1) fail	(3,1,1) fail	(4,1,1) pass
test configuration (1,2,1)				
No.	**Tuple Select**	**Generate Configurations**		
5	[-,2,1]	(2,2,1) fail	(3,2,1) fail	(4,2,1) fail
test configuration (2,1,1)				
No.	**Tuple Select**	**Generate Configurations**		
6	[2,1,-]	(2,1,2) fail	(2,1,3) fail	(2,1,4) fail
7	[2,-,-]	(2,2,2) pass	-	-
8	[2,-,1]	(2,3,1) pass	-	-
test configuration (3,1,1)				
No.	**Tuple Select**	**Generate Configurations**		
9	[3,1,-]	(2,2,1) fail	(3,2,1) fail	(4,2,1) fail
10	[3,-,-]	(3,2,2) pass	-	-
11	[3,-,1]	(3,3,1) pass	-	-

IV. EVALUATION

The goal of this section is to assess the performance of our appoach. Specifically, we present several experiments that are designed to answer the following questions:

Q1. Is our approach efficient compared with existing methods?

Q2. How effective does our approach deal with the overlapping faulty tuple?

Q3. How well does our approach deal with the newly introduced failure-inducing tuples?

Q4. Can our approach identify the failure-inducing combinations in the real software?

In the following experiments, we compare to an existing method, namely FIC_BS, which is stated to perform the best among the existing methods [4]. We set the variable *num_needed* of Algorithm 2 to 3.

A. Comparison with existing methods

To compare with method FIC_BS, we used five SUTs as our experimental subjects. The numbers of the parameters for these SUTs are 8, 9, 10, 11, and 12 respectively. For each SUT we conducted the following two groups of experiments:

1) In the first group of experiments, we assume that a failing test configuration of a SUT only contains a single t-size (t = 2, 3, 4) faulty tuple.

For example, the failing test configuration (1, 2, 2, 2, 1, 1, 2, 3) of the first SUT only contains a faulty tuple [1,2,-,-,-,-,-,-], other tuples in this configuration are healthy tuples. In fact, there are C_{m}^{t} (m = 8, 9, 10, 11, 12; t = 2, 3, 4) possible tuples of this kind in a failing configuration.

When embedding a single faulty tuple in a SUT, e.g., embedding a faulty tuple [1,1,-,-,-,-,-,-] in the first SUT, we give a failing test configuration as the input to the algorithm, for example: (1,1,1,1,1,1,1,1). To be fair, we do not give any information about other test configurations. Then we identify the embedded faulty tuple in the failing test configuration with our approach with TRT assumption (labeled as PATH), our augmented approach without TRT assumption (labeled as PATH _NA) and FIC_BS respectively. We record the extra test configurations they generated through the process of identifying the embedded faulty tuple.

We successively change the single faulty tuple embedded in the SUT from the C_{m}^{t} tuples and record the extra test configurations generated by the three algorithms in each iteration. Table VI lists the average extra configurations that have to be executed by each algorithm in the column titled "Average extra test configurations".

In addition to the average extra test configurations, there is another factor to check: the number of tuples in the TRT that are determined by each approach. Our two approaches—PATH and PATH_NA determined all the tuples in the TRT, but this is not the case for the FIC_BS. We should compute the number of the tuples in the TRT that are determined by FIC_BS. We list

the average covered number of tuples in TRT in the column with name "Average covered tuples" of Table VI.

We measure the efficiency of a method using the result obtained by dividing the number of extra test configurations by the coverage of tuples in the TRT. Table VI lists the average efficiency in the column with name "Efficiency".

2) In the second group of experiments, we assume that a failing test configuration of a SUT contains two t-size (t = 2, 3, 4) faulty tuples, and the two tuples do not overlap.

For example, assume that tuple (1, 2, 2, 2, 1, 1, 2, 3) of the first SUT contains two faulty tuples: [1,2,-,-,-,-,-,-] and [-,-,-,2,-,1,-,-]. These two tuples do not overlap.

When embedding two faulty tuples that do no overlap in a SUT, we identify the two faulty tuples with three approaches. As in the first group of experiments, we record the number of generated extra configurations, covered tuples, and efficiency.

We then successively change the two faulty tuples having no overlapped part embedded in the SUT. After running all the cases, we list the average of extra configurations, covered tuples and efficiency of each algorithm in table VII.

TABLE VI. COMPARING RESULT OF SINGLE FAULTY TUPLE

SUT	t	Average extra test configurations			Average covered tuples			Efficiency		
		PATH	PATH _NA	FIC_BS	PATH	PATH _NA	FIC_BS	PATH	PATH _NA	FIC_BS
1	2	10.8	23.8	9.0	255	255	163.3	23.7	10.7	18.2
	3	12.4	25.7	12.0	255	255	140.4	20.5	9.9	11.5
	4	12.7	23.9	14.9	255	255	147.5	20.1	10.7	9.7
2	2	11.3	24.7	9.6	511	511	318.8	45.1	20.7	33.2
	3	13.7	27.9	12.9	511	511	263.8	37.4	18.3	20.0
	4	14.9	28.8	16.1	511	511	269.8	34.3	17.7	16.4
3	2	11.8	26.2	9.9	1023	1023	624.6	86.7	39.0	63.1
	3	14.2	29.4	13.3	1023	1023	500.0	72.0	34.8	36.8
	4	16.0	31.4	16.7	1023	1023	498.4	64.0	32.5	29.0
4	2	13.2	30.1	10.3	2047	2047	1227.6	155.3	68.0	120.1
	3	15.2	32.3	13.8	2047	2047	954.5	134.2	63.4	68.0
	4	17.0	34.1	17.3	2047	2047	928.8	120.2	60.0	52.4
5	2	13.7	31.3	10.4	4095	4095	2419.2	299.6	130.7	232.4
	3	16.7	36.2	14.1	4095	4095	1832.6	246.7	113.0	127.9
	4	18.2	37.3	17.7	4095	4095	1743.6	225.0	109.7	96.0

TABLE VII. COMPARING RESULT OF TWO NON OVERLAPPING FAULTY TUPLES

SUT	t	Average extra test configurations			Average covered tuples			Efficiency		
		PATH	PATH _NA	FIC_BS	PATH	PATH _NA	FIC_BS	PATH	PATH _NA	FIC_BS
1	2	25.0	55.7	17.2	255	255	158.0	10.2	4.6	9.2
	3	34.0	66.8	21.7	255	255	119.5	7.5	3.8	5.5
	4	41.0	72.5	24.2	255	255	111.6	6.2	3.5	4.6
2	2	27.0	58.9	17.9	511	511	307.2	18.9	8.7	17.1
	3	39.6	78.4	23.3	511	511	221.7	12.9	6.5	9.4
	4	48.0	85.3	27.6	511	511	198.0	10.6	6.0	7.1
3	2	28.6	63.5	18.6	1023	1023	600.7	35.8	16.1	32.3
	3	42.6	85.0	24.4	1023	1023	416.4	24.0	12.0	17.0
	4	54.6	99.7	29.4	1023	1023	356.7	18.7	10.3	12.0
4	2	31.0	69.7	19.9	2047	2047	1179.5	65.9	29.4	59.6
	3	46.1	93.7	25.5	2047	2047	790.2	44.4	21.8	30.9
	4	59.1	110.2	31.0	2047	2047	651.4	34.6	18.6	20.8
5	2	34.0	77.7	20.3	4095	4095	2323.8	120.4	52.7	114.8
	3	49.3	101.5	26.9	4095	4095	1512.2	83.1	40.4	56.0
	4	63.5	120.5	32.1	4095	4095	1203.3	64.4	34.0	37.1

From the results in Table VI and Table VII, we get the followed conclusions:

Firstly, we can observe that the efficiency of the approaches (PATH, PATH_NA and FIC_BS) decreases with the increase

of *t* for each SUT. And the efficiency increases with the increase of the size, i.e., the number of parameters, of SUT.

What's more, comparing the results of Table VI and Table VII, we can see the efficiency of identifying the failure-inducing combinations of SUT with a single faulty tuple is higher than those with two faulty tuples, which means that more effort is needed to identify the additional faulty tuple in the failing test configuration.

Thirdly, we can observe that although the algorithm—FIC_BS needs smaller test configurations in most cases (there exists some cases where the FIC_BS approach requires more test configurations than our path approach, that is when t = 4, SUT from 1to 4 in the Table VI). Note that such cases only cover a small number of tuples in the TRT. As a result, the efficiency of FIC_BS is poorer than our first approach—PATH in all cases as shown in Table VI and Table VII.

Lastly, the efficiency of the augmented approach—PATH_NA is poorer than PATH and FIC_BS. This is because this approach needs more extra test configurations to determine a tuple than the other two approaches, so that it can achieve better performance under the condition when new faulty tuples may be introduced in the generated test configurations.

B. Handling Overlapping faulty tuples

In the following experiments, we use the same SUTs in the previous section. To see how well our approach handles the case when a failing test configuration contains multiple faulty tuples that overlap, we design the following experiment:

We assume that a failing test configuration contains two t-size (t = 2, 3, 4) faulty tuples, and the two tuples overlap.

For example, (1, 2, 2, 2, 1, 1, 2, 3) contain two faulty tuples: [1,2,2,-,-,-,-,-] and [-,-,2,2,-,1,-,-], the third parameter value of the first tuple and the first parameter value are the same.

When embedding two overlapping faulty tuples in a SUT, we identify the two embedded faulty tuples with the three approaches. We also record their generated average extra configurations, covered tuples in the TRT and the efficiency, similar to the previous experiments. In addition, we record the number of failure-inducing tuples that the three approaches identified in the column titled "failure-inducing tuples identified".

We successively change the two different faulty tuples embedded in the SUT. After running all the cases, we list the average number of extra configurations, covered tuples, efficiency and the number of identified failure-inducing tuples of each algorithm in Table VIII.

TABLE VIII. COMPARING RESULT OF TWO OVERLAPPING FAULTY TUPLES

SUT	t	Average extra test configurations			Average covered tuples in TRT			Efficiency			failure-inducing tuples identified		
		PATH	PATH _NA	FIC_BS	PATH	PATH _NA	FIC_BS	PATH	PATH _NA	FIC_BS	PATH	PATH _NA	FIC_BS
1	2	17.4	41.7	8.5	255	255	149.3	14.6	6.1	17.6	2	2	1
	3	22.6	48.5	11.4	255	255	113.5	11.3	5.3	9.9	2	2	1
	4	24.5	48.9	14.2	255	255	114.0	10.4	5.2	7.9	2	2	1
2	2	18.3	43.1	9.0	511	511	293.1	27.9	11.8	32.8	2	2	1
	3	26.2	56.4	12.1	511	511	212.6	19.5	9.1	17.3	2	2	1
	4	29.7	58.8	15.3	511	511	203.2	17.2	8.7	13.0	2	2	1
3	2	19.1	45.4	9.4	1023	1023	577.4	53.5	22.5	62.0	2	2	1
	3	28.0	60.4	12.6	1023	1023	402.6	36.5	16.9	31.7	2	2	1
	4	34.7	70.1	15.8	1023	1023	367.6	29.5	14.6	22.8	2	2	1
4	2	20.7	49.9	9.8	2047	2047	1140.6	98.9	41.0	118.0	2	2	1
	3	30.4	66.5	13.1	2047	2047	769.1	67.3	30.8	58.2	2	2	1
	4	38.6	78.6	16.5	2047	2047	673.9	53.1	26.1	40.1	2	2	1
5	2	22.4	54.5	10.0	4095	4095	2257.7	183.1	75.1	227.8	2	2	1
	3	32.8	72.6	13.4	4095	4095	1480.0	124.7	56.4	109.4	2	2	1
	4	42.1	86.7	16.9	4095	4095	1249.1	97.2	47.2	72.3	2	2	1

From the results in Table VIII, we find that our two approaches identified all the two failure-inducing tuples with overlap part while FIC_BS only found one faulty tuple of the two failure-inducing tuples in all cases.

From the previous section, we knew that the efficiency of identifying single faulty tuple is higher than identifying two faulty tuples. Importantly, even though our approach identified two faulty tuples while FIC_BS identified only one, our approach, PATH, still manage to achieve higher efficiency than FIC_BS in most cases (except the cases when *t* is 2).

Furthermore, compared with Table VII, we find the efficiency of identifying the overlapping tuples is higher than identifying the tuples that do not overlap. In other words, our approaches have a better performance at identifying failure-inducing tuples with overlapped part.

C. Handling newly introduced faulty tuples

We use the same five SUTs in the previous section. To see how well our approach handles the case where the generated extra test configurations could contain newly introduced failure-inducing tuples, we design the following experiment:

We inject two faulty tuples in the SUT. One must be contained in the test configuration under analysis; another must not be exist in the test configuration.

For example: for the test configuration (1, 1, 1, 1, 1, 1, 1, 1) of the first SUT, we inject a faulty tuple [1,1,-,-,-,-,-,-] which is contained in this test configuration, and another faulty tuple [-,2,2,-,-,-,-,-] which is not contained in the test configuration.

We use two-level, nested loops to generate two faulty tuples and identify them using three approaches. The first level layer loop is to successively select a t-size (t = 2,3,4) faulty tuple from the test configuration under analysis. In the second level loop, we first generate a random test configuration which is different from the original test configuration at each parameter, say, (3,2,2,2,3,2,2,3), and then successive select a 2-way faulty tuple from the newly generated test configuration. After two tuples are embedded in the SUT, we identify them using the three approaches, and then record the number of correctly identified faulty tuples which is contained in the test configuration. In addition, we recorded the number of newly introduced faulty tuples identified and the incorrectly identified faulty tuples, which are neither the first embedded faulty tuple nor the second embedded faulty tuple.

Table IX shows, for each approach, the total number of correctly identified faulty tuples, the total number of newly introduced faulty tuples identified and the total number of the incorrectly identified faulty tuples in columns named "correctly Identified tuples"," new imported faulty tuples Identified" and "incorrectly Identified tuples", respectively.

TABLE IX. RESULT OF IDENTIFYING NEW IMPORTED FAILURE-INDUCING TUPLE

SUT	t	correctly Identified tuples			new imported faulty tuples Identified			incorrectly Identified tuples		
		PATH	PATH _NA	FIC_BS	PATH	PATH _NA	FIC_BS	PATH	PATH _NA	FIC_BS
1	2	601	784	364	0	241	0	364	0	420
	3	1173	1568	728	0	618	0	478	0	840
	4	1736	1960	1260	0	297	0	245	0	700
2	2	1170	1296	1080	0	189	0	190	0	216
	3	2720	3024	2520	0	478	0	362	0	504
	4	3190	4536	1890	0	2396	0	1607	0	2646
3	2	1777	2025	1575	0	410	0	495	0	450
	3	4182	5400	2880	0	2053	0	1625	0	2520
	4	7569	9450	6300	0	3350	0	2156	0	3150
4	2	2688	3025	2475	0	710	0	913	0	550
	3	6691	9075	4455	0	4194	0	4455	0	4620
	4	12363	18150	8910	0	11466	0	7635	0	9240
5	2	3579	4356	2970	0	1252	0	1255	0	1386
	3	10663	14520	8360	0	7119	0	5298	0	6160
	4	23943	32670	18810	0	22370	0	14225	0	13860

The results in Table IX suggest that only our approach—PATH_NA can identify the newly introduced failure-inducing tuples. Furthermore, only PATH _NA did not identify faulty tuples incorrectly, which means that this approach is not affected by the newly introduced faulty tuples, while the other two approaches: PATH and FIC_BS are affected by the newly introduced faulty tuples.

D. Handling failure-inducing combinations in real software

We used a module of the Traffic Collision Avoidance System (TCAS) benchmark as our real software testing subject (available at [14]), The module is part of a set of C programs that has been used in other evaluations of software testing methods [13], [15].

The program has 12 input parameters and one output parameter. To make model checking feasible, we use the same partitioned equivalence classes of each parameter as [15] ,We can model this input configuration as SUT(3 , 2 , 2 , 2 , 2 , 2 , 4 , 10, 10, 3, 2, 2).

We get 5 faulty versions of the TCAS by manually seeding realistic faults into the correct version. And for a test configuration, the testing result is determined by comparing the executed result of the correct version and faulty version. We induce a failing test configuration as the input of each algorithm, and make analysis of the results obtained.

Before testing, we manually identify the actually failure-inducing combinations of each incorrectly version of TCAS through code inspection, which are those combinations of input configurations which will make the result of the corresponding incorrectly version of TCAS differ from the correct version.

The result is listed in table X.The column with the field name "Version" means the faulty verstion of the TCAS, and column "All" shows the actually number of failure-inducing combinations in this version of TCAS. The remaining columns show the results of each algorithm, the sub column with name "R" shows the number of correctly indentified failure-inducing combintions, and the sub column "W" shows the number of incorrectly identified combinations.

TABLE X. RESULT OF IDENTIFIED REALISTIC FAILURE-INDUCING COMBINATIONS IN REAL SOFTWARE

Version	All	PATH		PATH _NA		FIC_BS	
		R	W	R	W	R	W
1	2	2	0	2	0	1	0
2	2	1	1	2	0	1	0
3	1	1	0	1	0	1	0
4	3	0	1	3	0	1	0
5	4	2	0	2	0	0	1
Total	12	6	2	10	0	4	1

We find that our approach—PATH_NA get the best performance in identifying the realistic failure-induce combinations among the three approaches from table X.

V. RELATED WORKS

Nie's approach in [3] and [6] first separates the faulty-possible tuples and healthy-possible tuples into two sets. Subsequently, by changing a parameter value at a time of the original test configuration, this approach generates extra test configurations. After executing the configurations, the approach converges by reducing the number of tuples in the faulty-possible sets.

Delta debugging [5] proposed by Zeller is an adaptive divide–and-conquer approach to locating interaction fault. It is very efficient and has been applied to real software environment. Zhang et al. [4] also proposed a similar approach that can identify the failure-inducing combinations that has no overlapped part efficiently,

Colbourn and McClary [7] proposed a non-adaptive method. Their approach extends the covering array to the locating array to detect and locate interaction faults.

C. Martínez [8-9] proposed two adaptive algorithms. The first one needs safe value as their assumption and the second one remove the assumption when the number of values of each parameter is equal to 2. Their algorithms focus on identifying the faulty tuples that have no more than 2 parameters.

Ghandehari.etc [10] defines the suspiciousness of tuple and suspiciousness of the environment of a tuple. Based on this, they rank the possible tuples and generate the test configurations. Although their approach imposes minimal assumption, it does not ensure that the tuples ranked in the top are the faulty tuples.

Yilmaz [11] proposed a machine learning method to identify inducing combinations from a combinatorial testing set. They construct a classified tree to analyze the covering arrays and detect potential faulty combinations. Beside this, Fouché [12] and Shakya [13] made some improvements in identifying failure-inducing combinations based on Yilmaz's work.

VI. CONCLUSION AND FUTURE WORK

In this paper, we introduced a new model to identify failure-inducing combinations. It uses TRT to record all the tuples under test and their relationships. Our approach can identify the overlapping faulty tuples while at the same time generating accurate results. The augmented approach—PATH_NA can handle the newly introduced failure-inducing tuples very well. Our approach exhibits very good efficiency when identifying the faulty tuples in a failing test configuration.

However, our approach needs further improvement as it uses large space to record the TRT (We need record 2^n nodes in a TRT). We have done some improvement to largely reduce the space needed to identify the minimal failure-inducing combinations, which will be reported in our future publications. Also the best number of extra test configurations needed to determine a faulty tuple will be investigated as an important part of our future work. Furthermore, we will apply our method into more complex software environments (such as SOA architecture software).

REFERENCES

[1] Kuhn, D. Richard, and Michael J. Reilly. "An investigation of the applicability of design of experiments to software testing." *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*. IEEE, 2002.

[2] C. Nie and H. Leung, "A survey of combinatorial testing," ACM Computing Surveys (CSUR), vol. 43, p. 11, 2011.

[3] C. Nie and H. Leung, "The minimal failure-causing schema of combinatorial testing," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 20, p. 15, 2011.

[4] Z. Zhang and J. Zhang, "Characterizing failure-causing parameter interactions by adaptive testing," in Proceedings of the 2011 International Symposium on Software Testing and Analysis, 2011, pp. 331-341.

[5] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," Software Engineering, IEEE Transactions on, vol. 28, pp. 183-200, 2002.

[6] Baowen, X., Changhai, N., Liang, S and Huo-Wang, C. "A software failure debugging method based on combinatorial design approach for testing." *Chinese Journal of Computers*, 29.1(2006), 132-138.

[7] C. J. Colbourn and D. W. McClary, "Locating and detecting arrays for interaction faults," Journal of combinatorial optimization, vol. 15, pp. 17-48, 2008.

[8] C. Martínez, et al., "Algorithms to locate errors using covering arrays," LATIN 2008: Theoretical Informatics, 2008.

[9] C. Martínez, et al., "Locating errors using ELAs, covering arrays, and adaptive testing algorithms," SIAM Journal on Discrete Mathematics, vol. 23, p. 1776, 2009.

[10] L. S. G. Ghandehari, et al., "Identifying failure-inducing combinations in a combinatorial test set," in Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on, 2012, pp. 370-379."

[11] C. Yilmaz, et al., "Covering arrays for efficient fault characterization in complex configuration spaces," Software Engineering, IEEE Transactions on, vol. 32, pp. 20-34, 2006.

[12] S. Fouché, et al., "Incremental covering array failure characterization in large configuration spaces," in Proceedings of the eighteenth international symposium on Software testing and analysis, 2009, pp. 177-188.

[13] K. Shakya, et al., "Isolating Failure-Inducing Combinations in Combinatorial Testing using Test Augmentation and Classification," in Proc. International Workshop on Combinatorial Testing (CT 2012), ed. Montreal, Canada, 2012.

[14] Aristotle sample analysis programs. Available: http://pleuma.cc.gatech.edu/aristotle/Tools/Demo/preDefSrc/tcas/index.html

[15] D. R. Kuhn and V. Okum, "Pseudo-exhaustive testing for software," in Software Engineering Workshop, 2006. SEW'06. 30th Annual IEEE/NASA, 2006, pp. 153-158.